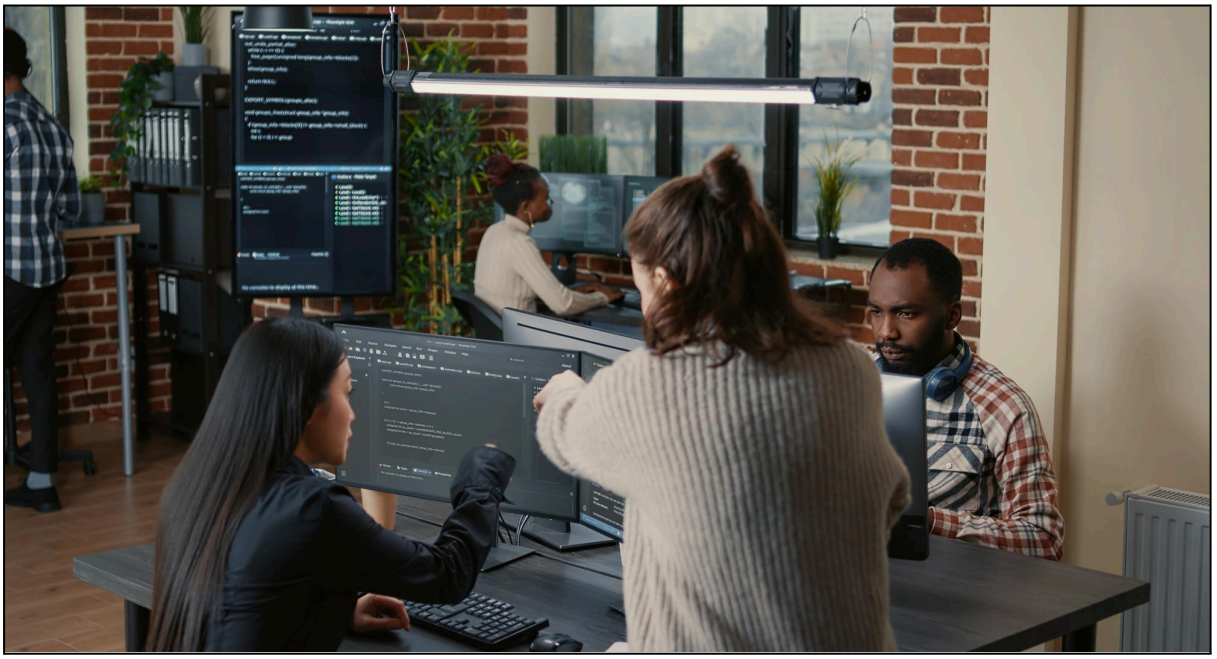
INSTITUTO FEDERAL
Santa Catarina

Formação continuada em Programação Front End com:

React + Javascript

Prof. Eduardo Gomes 2025/2



Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas:

Campus: Canoinhas

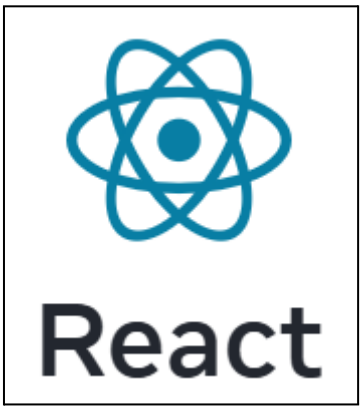
Índice

Introdução

O que é React?

O React é uma biblioteca JavaScript desenvolvida pelo Facebook, amplamente utilizada para a construção de interfaces de usuário dinâmicas e interativas conhecidas como Single Page Application (SPA).

Introduzida pela primeira vez em 2013, a biblioteca React tem como principal característica o conceito de "componente", que são blocos reutilizáveis que contêm partes da interface de usuário. Esses componentes permitem que os desenvolvedores construam aplicações de maneira modular e eficiente, facilitando tanto o desenvolvimento quanto a manutenção do código. A popularidade do React se deve também ao seu desempenho otimizado, possibilitado pela utilização do "Virtual DOM", que minimiza atualizações diretas no DOM real e melhora a experiência do usuário ao tornar as interfaces web mais rápidas e responsivas.



Logotipo oficial do React

Página e documentação oficial: https://react.dev/

O que é DOM e o que é Virtual DOM?

O **DOM** (Document Object Model) é uma representação em forma de árvore de todos os elementos HTML e seus atributos que compõem uma página web. Quando o navegador carrega uma página, ele transforma o HTML em um objeto que o JavaScript pode acessar e modificar dinamicamente — esse objeto é o DOM.

O **Virtual DOM**, por outro lado, é uma abstração do DOM real, criada na memória. Ele é uma das grandes inovações que o React trouxe. Quando o estado de um componente muda, o React primeiro atualiza esse DOM virtual — uma cópia leve e otimizada do DOM tradicional — e depois compara a nova versão com a anterior utilizando um processo chamado *reconciliation* (reconciliação). Ele calcula a diferença (chamada de *diffing*) e, em vez de redesenhar toda a árvore, atualiza apenas os nós que realmente mudaram no DOM real. Isso torna as atualizações muito mais rápidas e eficientes, melhorando significativamente a performance das aplicações, especialmente em interfaces com muitas interações e re-renderizações.

O que é Node.js?

Node.js é um ambiente de execução JavaScript (*Runtime*) que permite que você execute código JavaScript. Desenvolvido a partir do motor de JavaScript V8 do Google Chrome, Node.js é conhecido por sua capacidade de lidar com operações de entrada e saída de forma assíncrona e não bloqueante, o que significa que ele pode processar muitas conexões simultaneamente de forma eficaz. Isso o torna ideal para aplicações de rede escaláveis, como servidores web, APIs e aplicações em tempo real.

Página e documentação oficial: https://nodejs.org/pt

O que é NPM?

O npm, que significa ***Node Package Manager,*** é o gerenciador de pacotes padrão para o ambiente de execução Node.js. Ele permite que desenvolvedores instalem, compartilhem e gerenciem pacotes (módulos) de JavaScript que podem ser utilizados para estender as funcionalidades de suas aplicações. Ele também ajuda a gerenciar as dependências de um projeto, garantindo que todas as bibliotecas utilizadas estejam disponíveis e atualizadas conforme necessário. Através de um arquivo chamado package.json, você pode listar todas

as dependências necessárias para o seu projeto, facilitando a replicação do ambiente de desenvolvimento para outros membros da equipe.

O que é Vite?

Vite é uma ferramenta de **build** moderna para projetos front-end que oferece um ambiente de desenvolvimento extremamente rápido e eficiente. Criado por Evan You, o mesmo criador do Vue.js, Vite é projetado para substituir ferramentas de build mais tradicionais, oferecendo tempos de inicialização significativamente mais rápidos e uma experiência de desenvolvimento mais ágil.

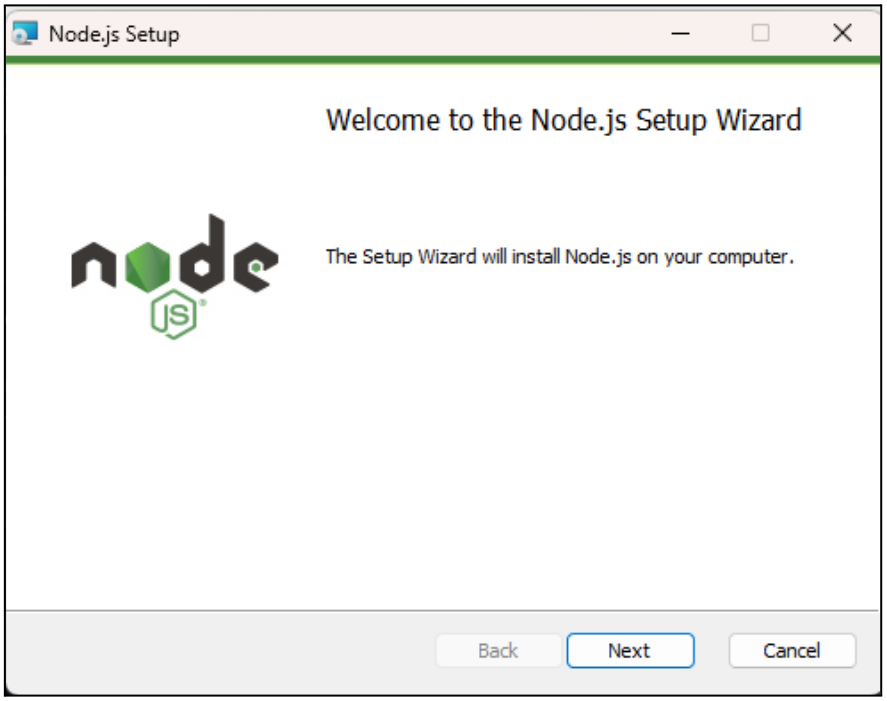
Instalando Node.js e Configurando o Ambiente

Instalação do Node.js

1. Acesse o Site Oficial do Node.js
    a. Navegue até o site oficial nodejs.org.

2. Baixe o Instalador (Clique em Descarregar)
    a. Na página principal, você verá duas versões para download: a LTS (Long Term Support) e a Current. Geralmente, a versão LTS é recomendada, pois é a mais estável.
3. Clique no botão de download da versão que preferir (provavelmente a LTS).

4. Execute o Instalador
    a. Vá até a pasta onde o arquivo foi baixado e clique duas vezes para executar o instalador e siga os passos de instalação até o final.



Verifique se o Node.js está instalado corretamente:

No Windows:

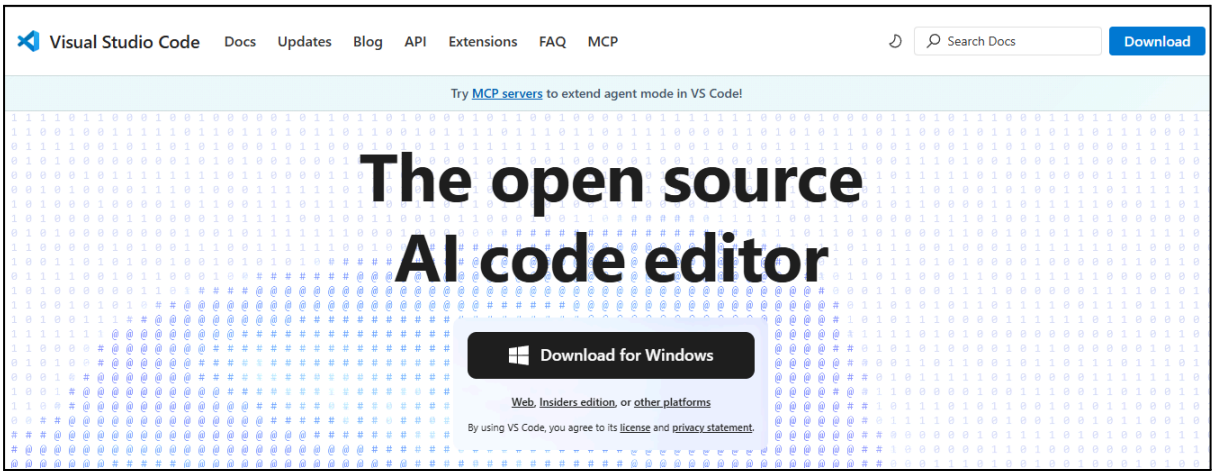
- Abra o Prompt de Comando (pressione `Windows + R`, digite cmd e pressione Enter).
- Digite `node -v` para verificar a versão do Node.js instalada.
- Digite `npm -v` para verificar a versão do npm instalada.

Configurando um Editor de Código

Para escrever e executar nossos códigos, precisamos de um editor ou ambiente de desenvolvimento. Vamos usar o **Visual Studio Code (VS Code)**, que é gratuito e fácil de usar.

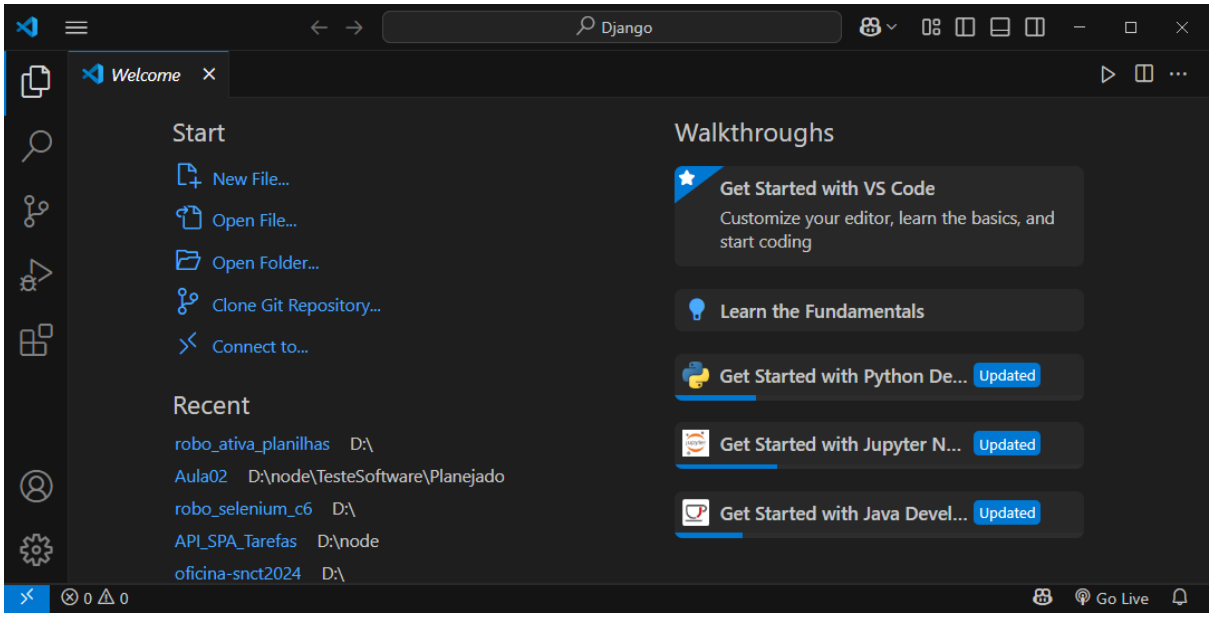
1. Baixando o VS Code

- Acesse: https://code.visualstudio.com/
- Clique em **Download** e escolha a versão para o seu sistema operacional.



2. Instalando o VS Code

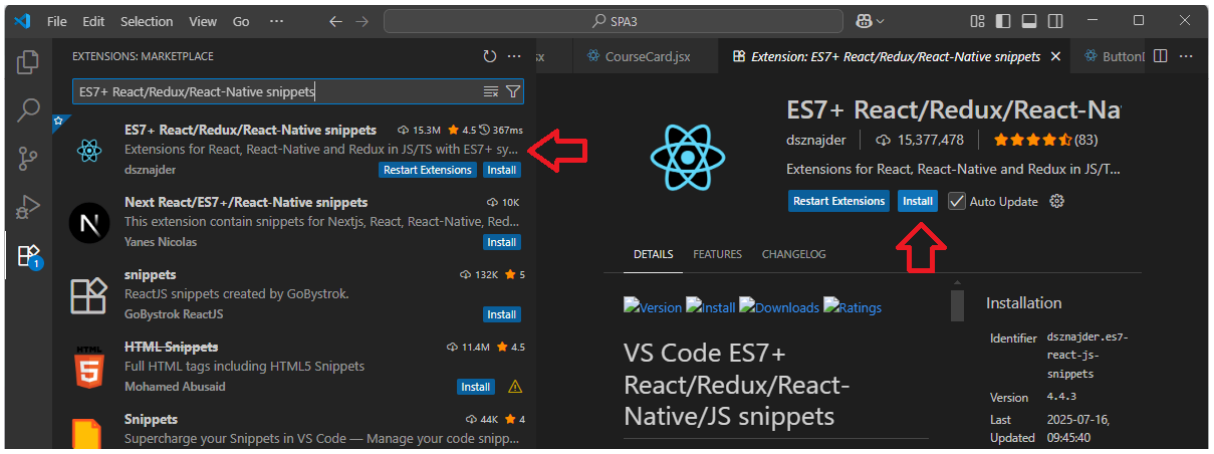
- **No Windows**
  - Execute o instalador baixado e siga as instruções padrão.



Tela inicial

3. Configurando o VS Code para React

- Abra o VS Code.
- Vá para a aba **Extensões** (ícone de quadrados à esquerda ou Ctrl+Shift+X).
- Pesquise por **ES7+ React/Redux/React-Native snippets**.
- Instale a extensão.



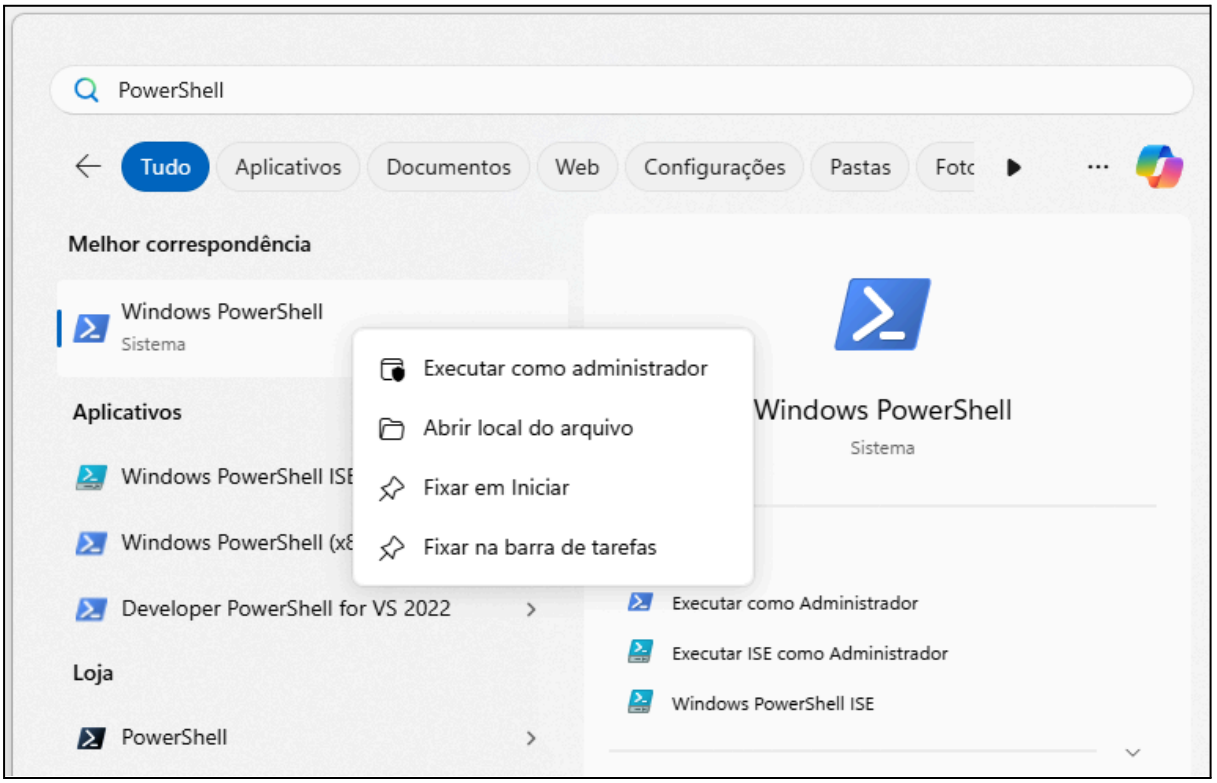
Os passos para instalar a extensão

4. Permissão de execução de scripts no Powershell

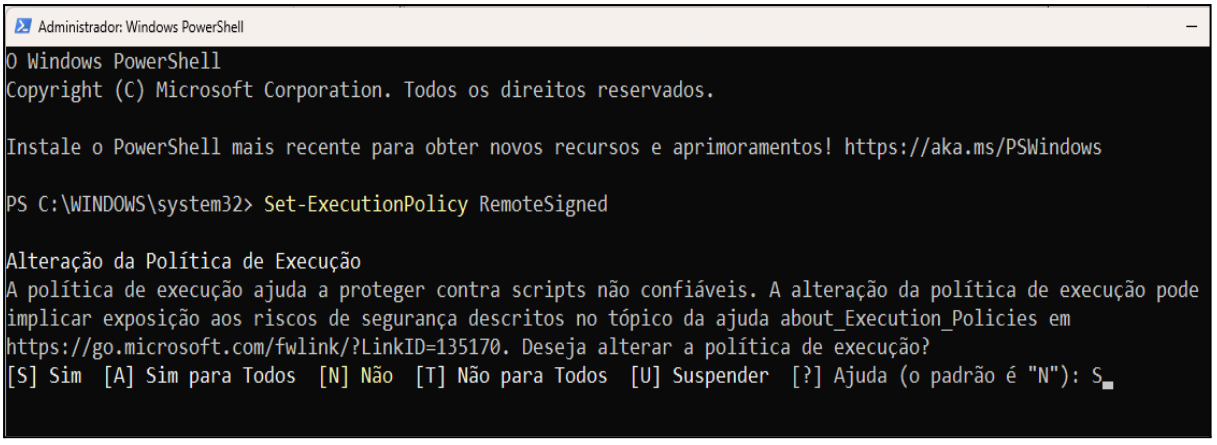
Para usuários do Windows 10 ou superior, por padrão o Powershell vem configurado para não permitir execução de scripts. Para liberar a execução realize os seguintes passos:

- Abra o Powershell como administrador;

- Digite PowerShell na pesquisa de aplicativos, clique com o botão esquerdo e em **Executar como administrador**.



- Digite o comando: `Set-ExecutionPolicy RemoteSigned`
- Responda "S" para a pergunta: `Deseja alterar a política de execução?`



- Execute o comando `Get-ExecutionPolicy -List` para conferir se alterou corretamente.
- A linha `LocalMachine` deve estar como: `RemoteSigned.`

```
PS C:\WINDOWS\system32> Get-ExecutionPolicy -List

        Scope ExecutionPolicy
        ----- ---------------
MachinePolicy       Undefined
   UserPolicy       Undefined
      Process       Undefined
  CurrentUser       Undefined
 LocalMachine    RemoteSigned


PS C:\WINDOWS\system32> _
```

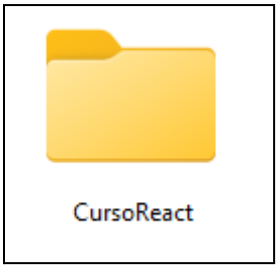
- Digite `node -v` para verificar a versão do Node.js instalada.
- Digite `npm -v` para verificar a versão do npm instalada.

Criando o Primeiro Projeto - Hello World

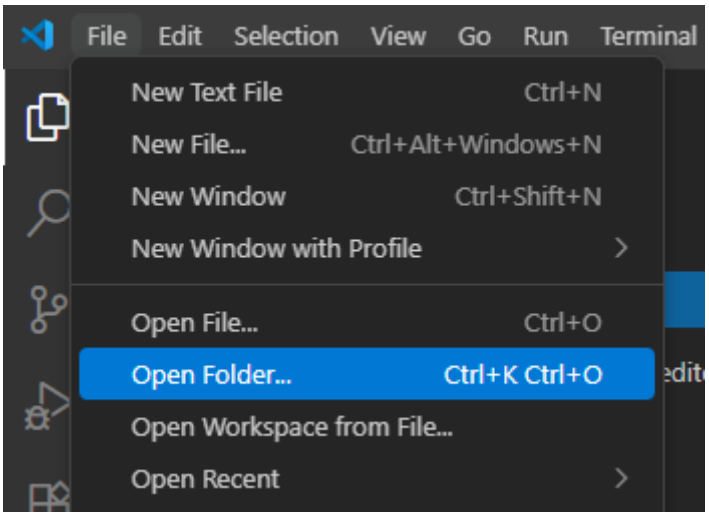
Para criar o primeiro projeto, vamos utilizar a ferramenta de *build* `Vite`. Siga os seguintes passos:

1. Criação da Pasta e Acesso pelo VS Code

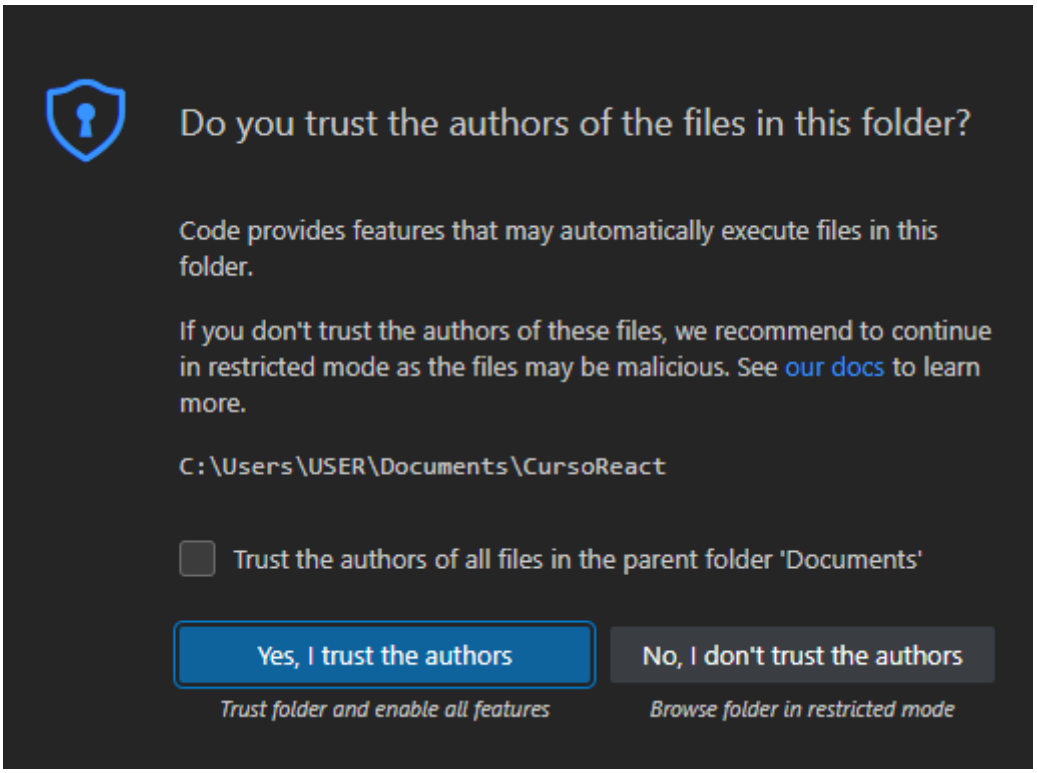
- Crie uma pasta onde deseja salvar seus projetos a chame de `CursoReact`.



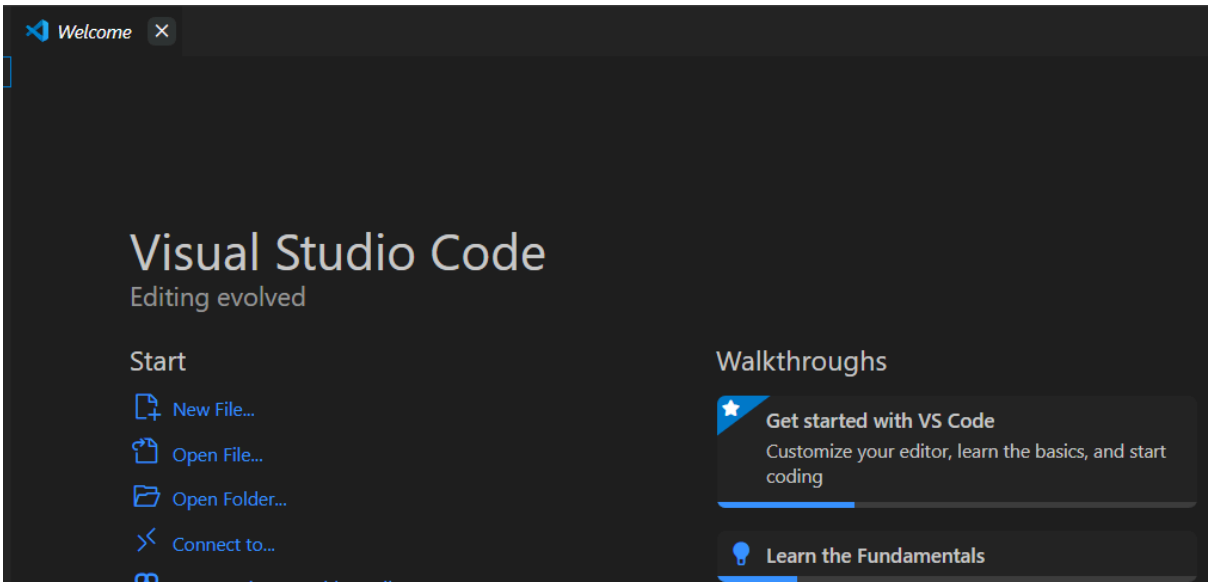
- Abra o VS Code e acesse: `File->Open Folder`



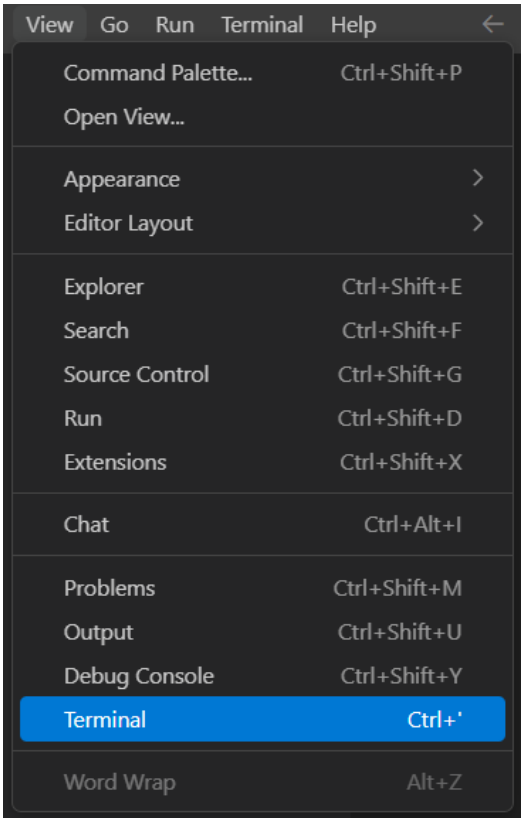
- Localize a pasta `CursoReact` e clique em **Selecionar Pasta**.
- O VS Code vai perguntar se você confia nos autores dos arquivos existentes na pasta. Clique em **Yes, I trust the autors**.



- Feche a aba Welcome.



- Acesse o menu `View -> Terminal`.



2. Criação do App usando o Vite

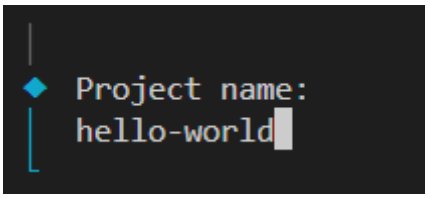
- No terminal digite o seguinte comando:

```
npm create vite@latest
```

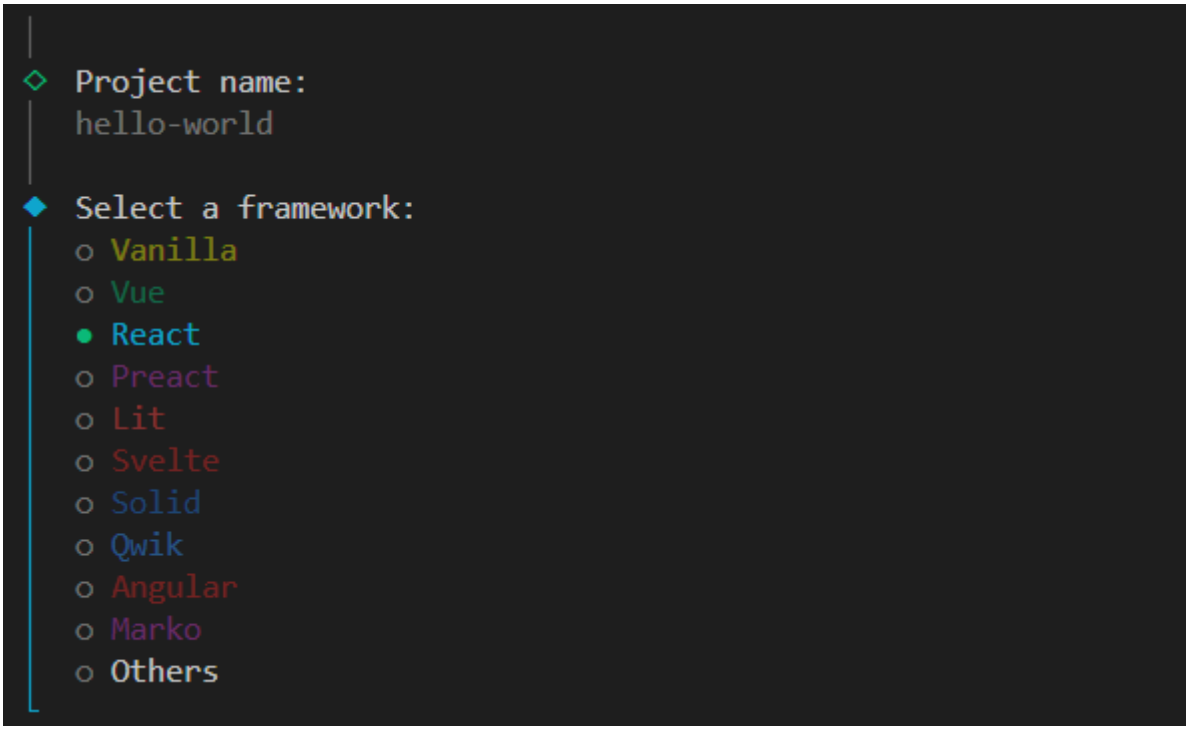
- Digite `y` para prosseguir.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\USER\Documents\CursoReact> npm create vite@latest
Need to install the following packages:
create-vite@7.1.0
Ok to proceed? (y) y
```

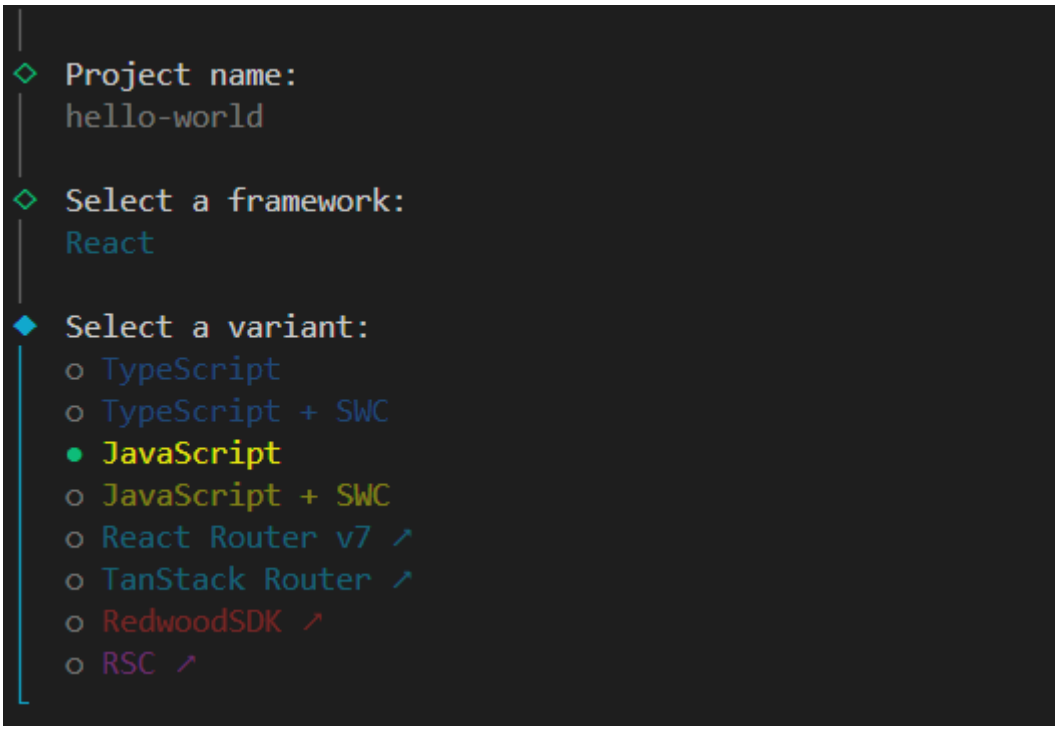
- Digite o nome do projeto: `hello-world`



- Com as setas selecione o framework `React` e pressione `<Enter>`

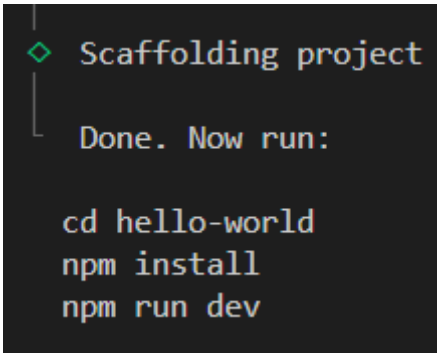


- Com as setas selecione a *variant* `Javascript` e pressione `<Enter>`

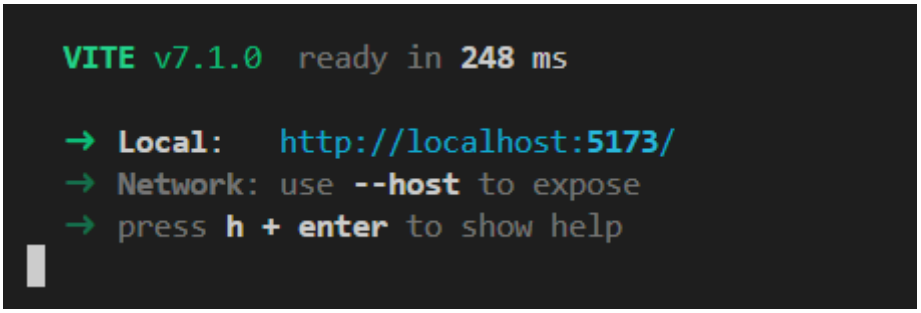


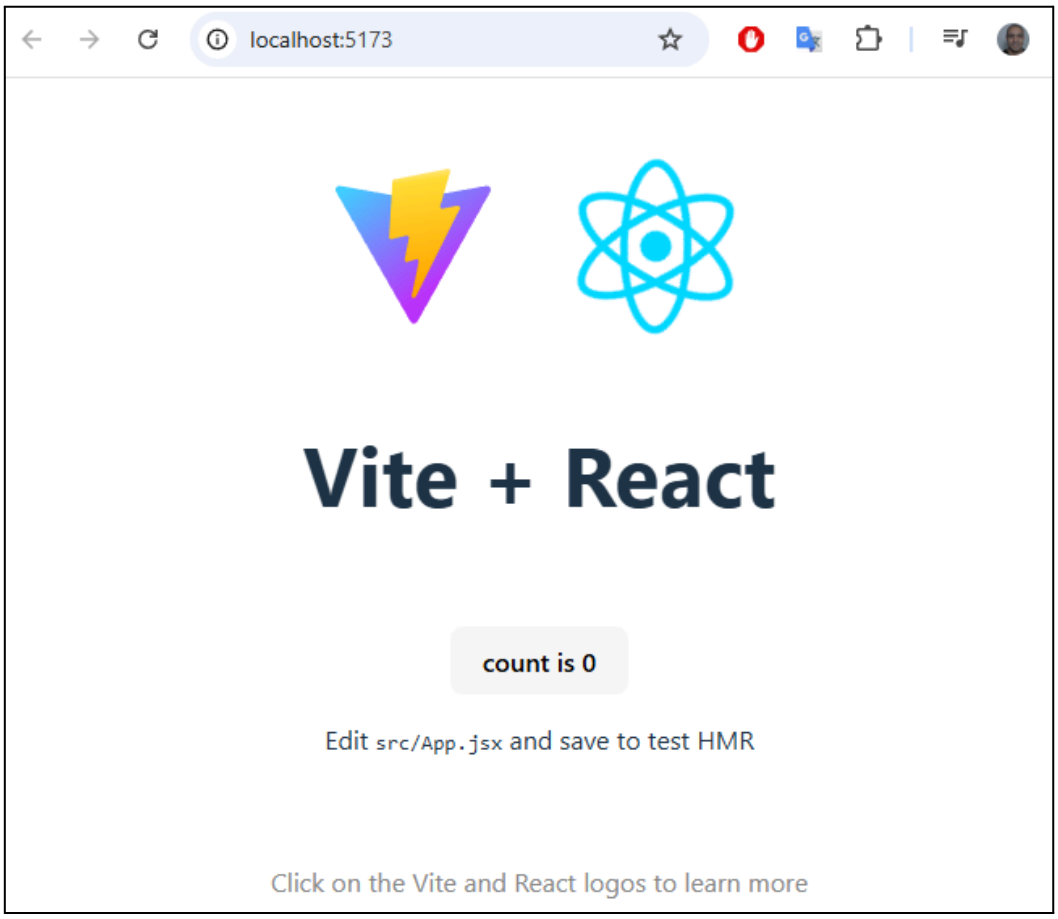
Pronto! O projeto está criado e o próprio Vite já dá as próximas instruções:

`cd hello-world` (Entra na pasta do projeto).
`npm install` (Instala as dependências - **precisa estar conectado na internet**).
`npm run dev` (Executa o projeto no template inicial).



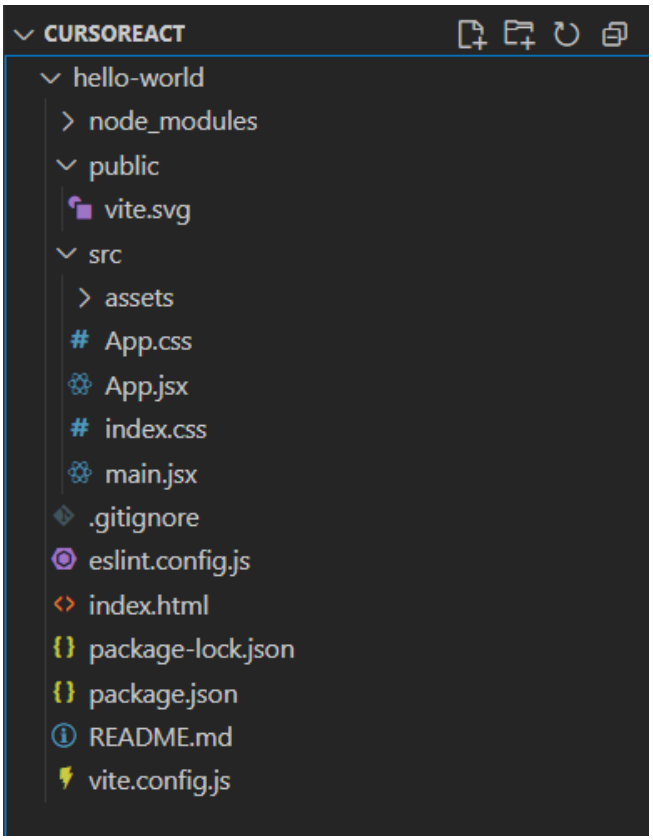
Após o comando `npm run dev` você já pode abrir o navegador para visualizar o template inicial criado pelo Vite.





Entendendo a estrutura de pastas

O Vite cria a seguinte estrutura de pastas para o nosso projeto:



Segue uma breve descrição para cada uma delas:

- **node_modules:** Este diretório contém todas as dependências do projeto gerenciadas pelo npm. Após instalar os pacotes especificados no package.json, o npm armazena os arquivos necessários aqui.
- **public:**
  - `vite.svg`**:** Localizado dentro do diretório **public**, este é um exemplo de arquivo público. Todo o conteúdo dessa pasta é copiado diretamente para a pasta de build final, sendo acessível pela raiz do domínio do seu site. É ideal para ícones, imagens ou outros ativos estáticos que não vão mudar com a publicação da aplicação.
- **src:**
  - `assets`**:** Um diretório dedicado para armazenar fontes, imagens, ou outros ativos usados na aplicação.
  - `App.css & index.css`**:** Estes arquivos contêm estilo CSS que pode ser aplicado globalmente ou a componentes específicos.
  - `App.jsx`**:** O componente **principal** de `React`, que serve como ponto de entrada da sua aplicação.

- **`main.jsx`**: O ponto de início do `React`. Geralmente, este arquivo é responsável por renderizar o componente **App** na DOM.

- **Arquivos de Configuração e Metadados:**
  - **`.gitignore`**: Contém uma lista de arquivos e diretórios que devem ser ignorados pelo controle de versão Git.
  - **`eslint.config.js`**: Configurações de linting, que ajudam a manter a consistência e a qualidade do código.
  - **`index.html`**: O arquivo HTML principal que serve como ponto de entrada para a aplicação. Este arquivo liga as partes essenciais do React e do Vite.
  - **`package.json`**: Define as dependências do projeto, scripts de execução e metadados como nome e versão.
  - **`package-lock.json`**: Garante instalações consistentes de pacotes, registrando versões exatas de dependências instaladas.
  - **`README.md`**: Um arquivo de documentação com informações básicas sobre o projeto.
  - **`vite.config.js`**: O arquivo de configuração do Vite, que personaliza o comportamento do servidor de desenvolvimento e o processo de build.

A melhor forma de aprender com este curso

Agora que já preparamos o ambiente e criamos nosso primeiro projeto, seguem dicas de como aproveitar melhor este curso:

- **Sempre** programe os códigos das aulas!
  - Dica: Assista primeiro, execute depois;
- O curso foi planejado **sequencialmente**, se não **domina** o React não pule aulas, especialmente para os projetos;
- **Faça** todos os exercícios propostos;
- Tire as suas **dúvidas** através do email do professor.

Fundamentos do React

A partir desta seção vamos aprender conceitos fundamentais para a criação de aplicações utilizando a biblioteca `React` com a linguagem de programação `Javascript`.

O que é JSX?

`JSX`, que significa `JavaScript XML`, é uma extensão de sintaxe para JavaScript. Com uma aparência similar ao `HTML`, o `JSX` permite que desenvolvedores descrevam a estrutura da interface do usuário diretamente dentro do código `Javascript`, facilitando a visualização de como a interface do usuário (UI) será composta. Essa integração com `Javascript` torna a construção de componentes mais intuitiva e expressiva.

O uso do `JSX` melhora a clareza e a legibilidade do código. Ao possibilitar a mistura de lógica `Javascript` com a estrutura da interface de forma coesa, o `JSX` não apenas simplifica o desenvolvimento, mas também promove uma manutenção mais eficiente do código. Dessa forma, ele se torna uma poderosa ferramenta no ecossistema `React`, contribuindo significativamente para o sucesso das `Single Page Applications (SPA)` que são as aplicações com interfaces de usuário dinâmicas e interativas.

Características do JSX e diferenças em relação ao HTML

Legibilidade:

Facilita a visualização de qual lógica está associada a cada parte da UI ao colocar a marcação diretamente no código.

Reutilização e Encapsulamento:

Permite criar componentes reutilizáveis, unindo lógica e marcação em um só lugar.

Diferenças Chave Entre JSX e HTML:

- Integração com JavaScript, pois expressões JavaScript podem ser inseridas diretamente usando `{}`.
- Para declaração de classes usa `className` em vez de `class` para classes CSS, pois `class` é uma palavra reservada no JavaScript.

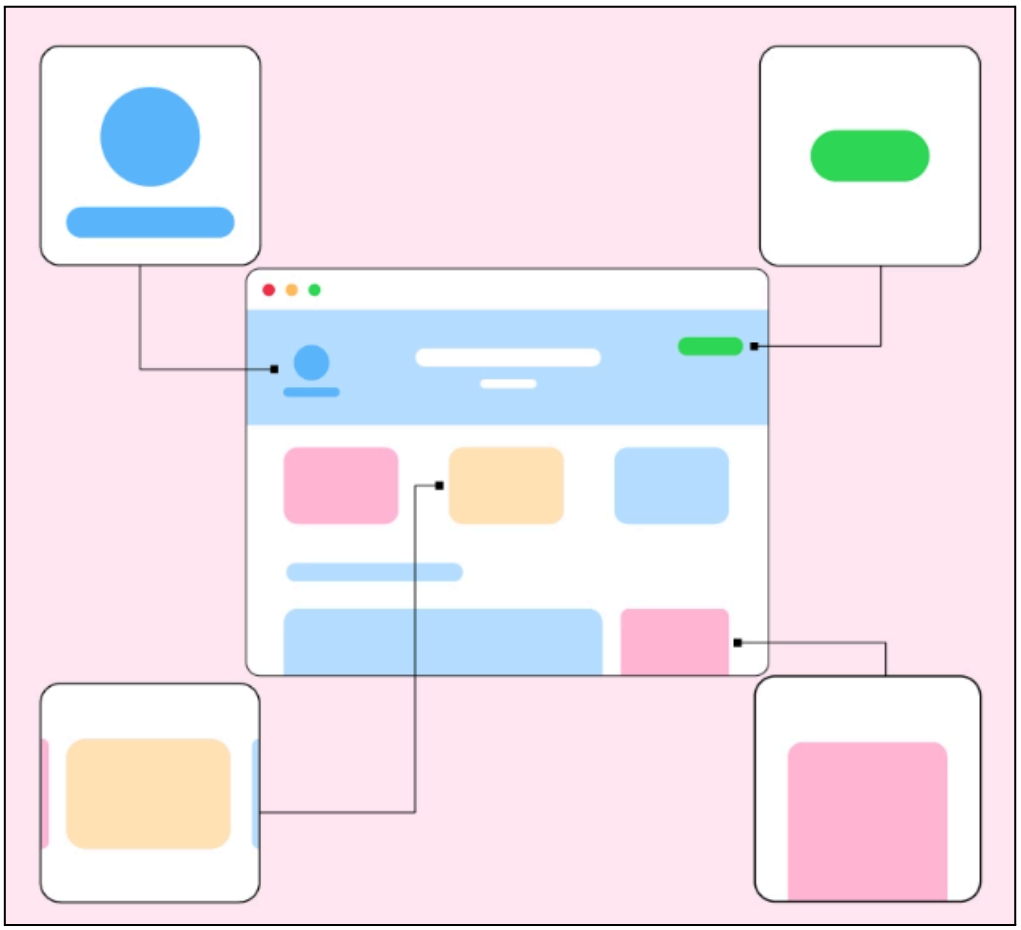
- Auto-fechamento, pois todos os elementos devem ser fechados; use `/` para auto-fechamento em elementos sem filhos, como `<img src="image.jpg" />`.
- Geralmente Tags HTML são escritas em minúsculas; componentes personalizados em JSX devem ser capitalizados (Primeira Letra Maiúscula) para diferenciar dos elementos nativos.

O que é são os componentes React?

Um dos conceitos centrais do React é a **componentização**, que envolve a decomposição da interface do usuário em componentes independentes e reutilizáveis, cada um projetado para uma única função. Os componentes são unidades de código que retornam elementos JSX, representando uma parte específica da interface.

Os componentes no React oferecem várias vantagens essenciais:

- Promovem a reutilização de código, permitindo que desenvolvedores criem interfaces de usuário de maneira mais eficiente e consistente.
- Encapsulam lógica e apresentação, resultando em melhor organização e maior previsibilidade do comportamento da aplicação.
- A manutenção se torna mais fácil, pois alterações em um componente específico não afetam outras partes da aplicação.
- Facilitam a colaboração entre equipes, pois dividem a aplicação em blocos de construção gerenciáveis, tornando o desenvolvimento mais modular e escalável.
- Possuem capacidade de gerenciar seu próprio estado, contribuindo significativamente para o desempenho e a flexibilidade das SPAs.



A imagem acima representa 4 componentes criados separadamente, e no centro da imagem, os componentes sendo utilizados na aplicação.

O código abaixo apresenta um exemplo da criação de um componente React.

```
import React from 'react'

function PrimeiroComponente() {
  return (
    <div>
        <h1>Meu primeiro componente React</h1>
        <p>Este é um exemplo de componente React</p>
    </div>
  )
}

export default PrimeiroComponente
```

Como colocar comentários no script JSX?

Comentários são utilizados para melhorar a legibilidade do código, explicando seu propósito ou lógica. Eles são essenciais para a colaboração entre desenvolvedores, pois clarificam partes complexas e documentam decisões de projeto. Para fazer um comentário em JSX, você usa a sintaxe de comentários em JavaScript, mas deve envolvê-los em chaves.

Para um comentário de linha única, você escreve:

`{/* seu comentário aqui */}`.

Para comentários de múltiplas linhas, a estrutura se mantém similar:

```
{/* Este é um comentário

de múltiplas linhas em JSX */}
```

Essa abordagem garante que os comentários não interfiram com o HTML dentro do JSX.

Código Javascript dentro do JSX (Template Expressions)

Dentro de JSX, você pode incorporar código JavaScript usando Template Expressions, que permitem a execução de expressões embutidas diretamente na marcação. Isso é feito envolvendo o código entre chaves {}. Essa funcionalidade proporciona uma grande flexibilidade ao integrar lógica diretamente na interface do usuário. Por exemplo, você pode exibir variáveis, executar funções ou até realizar operações matemáticas diretamente no JSX. As Template Expressions tornam o JSX mais dinâmico e interativo, permitindo que desenvolvedores adaptem a interface às mudanças de estado ou propriedades de maneira eficiente e intuitiva. Essa capacidade de misturar lógica e apresentação no mesmo local é um dos pontos fortes do React, tornando o desenvolvimento mais fluido e consistente.

Exemplo de componente com Template Expressions.

```
import React from 'react'

function PrimeiroComponente() {
    const nome = "Eduardo Gomes";
    const a = 10;
    const b = 5;
  return (
```

```
    <div>
        <h1>Meu primeiro componente React</h1>
        <p>Este é um exemplo de componente React</p>
        <p>Nome do usuário: {nome}, o resultado da soma é: {a+b}</p>
    </div>
  )
}

export default PrimeiroComponente
```

Eventos no Front End

No desenvolvimento frontend com React, eventos são usados para capturar interações do usuário e responder a elas dinamicamente. React fornece uma sintaxe limpa para lidar com eventos de maneira semelhante ao HTML convencional, mas com a conveniência do Javascript moderno. Por exemplo, eventos são escritos em **camelCase** (primeiraLetraMinuscula e demaisPrimeirasLetrasMaiúsculas semEspaços) e as funções de manipulação são passadas diretamente. Alguns dos eventos mais comuns em React incluem:

- **onClick:** Disparado quando um elemento é clicado.
- **onChange:** Executado quando o valor de um elemento, como um `<input>`, é alterado.
- **onSubmit:** Acionado quando um formulário é enviado.
- **onMouseEnter:** Disparado ao passar o mouse sobre um elemento.
- **onFocus:** Acionado quando um elemento recebe foco, como ao clicar em um campo de texto.
- **onBlur:** Executado quando um elemento perde o foco.

Esses eventos permitem criar interfaces de usuário interativas, aprimorando a experiência do usuário final.

Exemplo do uso de eventos:

```
import './App.css'

function App() {

  function handleClick() {
    alert("Olá, você clicou no botão Clique Aqui!");
  }
  return (
    <>
      <h1>Hello World!</h1>
```

```
      <p>Bem vindo à minha página</p>
      <div>
        <button onClick={handleClick}>Clique Aqui!</button>
      </div>
    </>
  )
}

export default App
```

Funções de renderização

No React, funções de renderização são utilizadas para retornar elementos JSX condicionalmente, permitindo um controle mais dinâmico sobre o que é exibido baseado no estado ou nas propriedades de um componente. Você pode escrever JSX diretamente em uma função, e ao chamá-la no corpo do componente, controlar o que será renderizado. Isso torna o código mais organizado e facilita o gerenciamento das interfaces de usuário complexas. Veja um exemplo simples de uso de uma condicional em um componente para manipular a renderização:

```
function Renderizacao() {
    function renderizarConteudo(x) {
        if (x) {
            return <h2>Verdadeiro! Posso renderizar isso</h2>;
        } else {
            return <h2>Falso! Posso renderizar aquilo</h2>;
        }
    }
    return (
        <div>
            {renderizarConteudo(true)}
            {renderizarConteudo(false)}
        </div>
    )
}

export default Renderizacao
```

Imagens em React

Em React, imagens podem ser geridas de diferentes maneiras dependendo de suas fontes e do modo como são utilizadas no projeto. Imagens **públicas** são colocadas na pasta `public`, tornando-as diretamente acessíveis via URL relativa, e são úteis para arquivos que

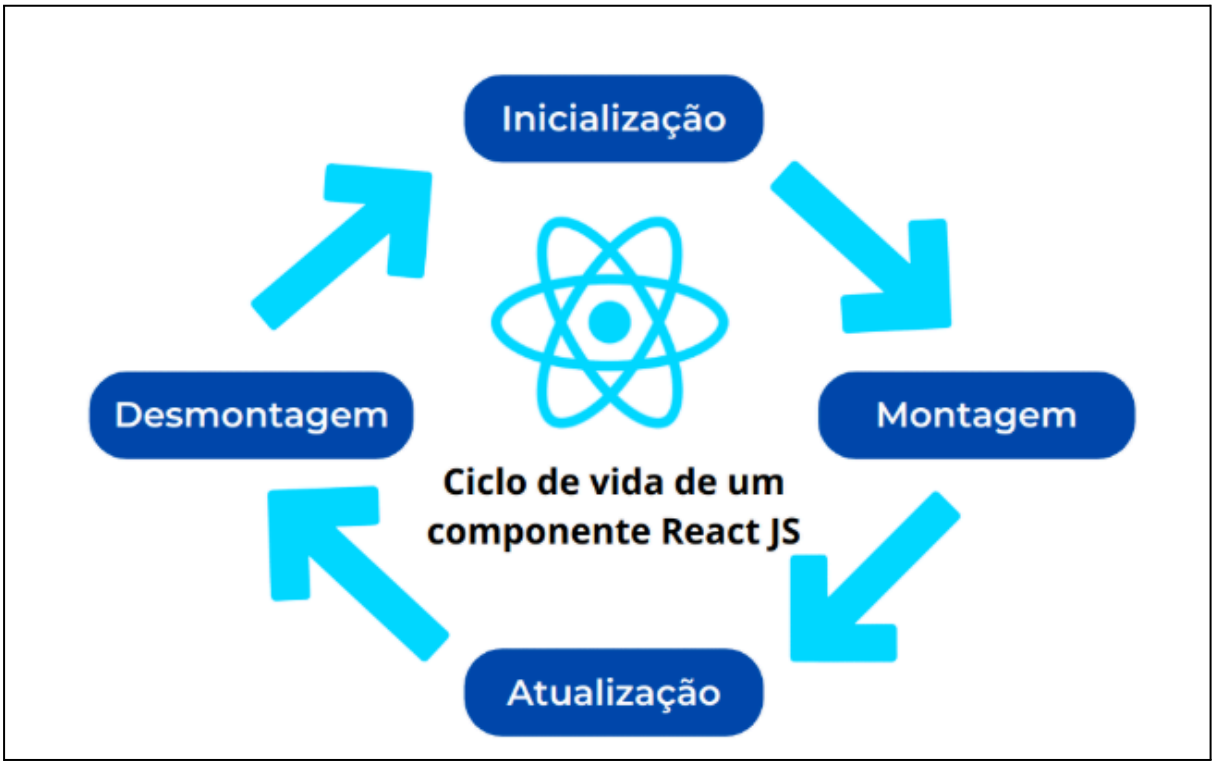
não mudam ou para ícones acessíveis globalmente. Já imagens dentro da pasta `src` são importadas no componente e fazem parte do processo de `build` (construção da aplicação), beneficiando-se de otimizações automáticas como compressão. Usar imagens na pasta `src` é ideal para imagens que estão diretamente associadas à lógica do componente. A escolha entre os dois depende da conveniência e necessidade da aplicação, com `public` oferecendo simplicidade na inclusão direta e `src` proporcionando melhor controle e otimização. Veja o exemplo de um componente que utiliza as duas formas:

```
import LogoSrc from './assets/logo.png';

function ImagensReact() {
  return (
    <div>
      <h1>Exemplo de Imagens em React</h1>
      <div>
        <h2>Imagem de Public</h2>
        <img src="image.png" width="100px" alt="Public Image" />
      </div>
      <div>
        <h2>Imagem de Src</h2>
        <img src={LogoSrc} width="100px" alt="Src Image" />
      </div>
    </div>
  );
}

export default ImagensReact;
```

Ciclo de Vida e Estado de um Componente React

Ciclo de Vida refere-se às etapas pelas quais um componente passa desde sua criação até sua remoção na página da aplicação. Em React, isso inclui fases como inicialização, montagem, atualização e desmontagem.

Estado é uma propriedade que componentes React usam para "lembrar" informações entre renderizações (similar a variáveis). O estado permite que um componente rastreie e reaja a mudanças de dados ao longo de sua existência, como o texto em um botão ou o conteúdo de um formulário.



Hooks

De uma forma bem resumida, **Hooks** são funções que facilitam o uso de recursos avançados em componentes funcionais no React. Eles permitem que você armazene informações, chamadas de **estados**, e execute ações em momentos específicos durante o **ciclo de vida** de um componente, como quando ele aparece ou desaparece na tela.

Hook useState

O hook `useState` é uma das funcionalidades mais utilizadas em React para gerenciar o estado em componentes. Ele permite que você adicione **variáveis de estado** ao seu componente, que podem ser atualizadas e utilizadas para re-renderizar a interface conforme necessário. Quando você chama `useState`, ele retorna um par de valores: o **estado atual** e uma **função** para atualizá-lo. Isso simplifica o processo de tornar sua interface interativa em resposta a eventos de usuários. Veja o exemplo abaixo:

```
import React, { useState } from 'react';

function Contador() {
  const [contador, setContador] = useState(0);
```

```
  return (
    <div>
      <p>Você clicou {contador} vezes</p>
      <button onClick={() => setContador(contador + 1)}>
        Clique aqui
      </button>
    </div>
  );
}
export default Contador;
```

Neste exemplo, é declarado um estado chamado contador, e toda vez que o usuário clica no botão, o estado é atualizado. Consequentemente, o React re-renderiza o elemento associado ao estado. Se fossem usadas variáveis simples, o elemento não seria re-renderizado, e o usuário não saberia quantas vezes o botão foi clicado.

Exemplo Calculadora de Médias

Abaixo um exemplo de um componente que recebe dois valores e calcula a média. O componente usa o **hook** `useState` para gerenciar as informações.

```
// Importa o React e o hook useState para gerenciar estados no componente
import React, { useState } from 'react';

// Importa o arquivo de estilos CSS para personalização do layout
import './CalculadoraMedia.css';

// Declaração do componente funcional CalculadoraMedia
function CalculadoraMedia() {

  // Estado para armazenar o valor do primeiro número (inicialmente vazio)
  const [numero1, setNumero1] = useState('');
  // Estado para armazenar o valor do segundo número (inicialmente vazio)
  const [numero2, setNumero2] = useState('');
  // Estado para armazenar a média calculada (inicialmente 0)
  const [media, setMedia] = useState(0);

  // Função para calcular a média
  const calcularMedia = () => {
     // Converte numero1 e numero2 para número decimal (float), soma, divide
por 2 e limita a 2 casas decimais
      const mediaCalculada = ((parseFloat(numero1) + parseFloat(numero2)) /
2).toFixed(2);

    // Atualiza o estado da média com o valor calculado
    setMedia(mediaCalculada);
  };

  // Estrutura JSX que será renderizada no navegador
```

```
  return (
    // Container principal com classe CSS para estilização
    <div className="calculadora-container">
      {/* Título da aplicação */}
      <h2>Calculadora de Média</h2>
      {/* Grupo do primeiro campo de entrada */}
      <div className="input-group">
        <label>
          Número 1:
          {/* Campo para digitar o primeiro número */}
          <input
            type="number" // Aceita apenas números
            autoFocus     // O cursor já inicia neste campo
            value={numero1} // Valor controlado pelo estado numero1
             onChange={(e) => setNumero1(e.target.value)} // Atualiza o estado
quando o usuário digita
          />
        </label>
      </div>
      {/* Grupo do segundo campo de entrada */}
      <div className="input-group">
        <label>
          Número 2:
          {/* Campo para digitar o segundo número */}
          <input
            type="number" // Aceita apenas números
            value={numero2} // Valor controlado pelo estado numero2
             onChange={(e) => setNumero2(e.target.value)} // Atualiza o estado
quando o usuário digita
          />
        </label>
      </div>
      {/* Botão que dispara o cálculo da média */}
      <button onClick={calcularMedia} className="calcular-button">
        Calcular Média
      </button>

      {/* Exibe o resultado da média */}
      <h2>Média: {media}</h2>
    </div>
  );
}
// Exporta o componente para que possa ser usado em outros arquivos
export default CalculadoraMedia;
```

Exercício: Calculadora de Volume da Piscina

Crie um componente React funcional chamado `CalculadoraVolume` que permita ao usuário calcular o volume de uma piscina. O componente deve:

- Importar o React e o hook `useState` para gerenciar o estado do componente.
- Incluir campos de entrada para:
  - Comprimento da piscina (em metros)
  - Largura da piscina (em metros)
  - Profundidade da piscina (em metros)
- Usar o estado para armazenar os valores dos inputs e o volume calculado.
- Implementar uma função que calcule o volume da piscina utilizando a fórmula:
  - `Volume = comprimento x largura x profundidade`
- Renderizar um botão que, ao ser clicado, execute a função de cálculo e atualize o estado do volume.
- Exibir o resultado do volume calculado na interface do usuário, com duas casas decimais.

CSS em React

Existem diversas formas de aplicarmos estilos CSS em nossos projetos React, abaixo apresento as formas mais comuns.

CSS Global no index.css

Em React, o CSS global é gerenciado frequentemente através de um arquivo index.css, importado no ponto de entrada da aplicação. Isso permite estilizar de modo consistente elementos que são comuns a toda a aplicação. A desvantagem é que o escopo global pode ocasionar conflitos de estilos se não houver cuidado com o nome das classes. Abaixo um exemplo de estilo no arquivo `index.css` que será aplicado para todo o projeto.

```
/* index.css */
body {
  margin: 0;
  font-family: Arial, sans-serif;
}

h1 {
  color: navy;
}
```

Este arquivo é importado na página de entrada da aplicação (`main.jsx`), conforme código abaixo:

```
/*main.jsx */
import './index.css'
```

CSS de Componente

Cada componente pode ter um arquivo CSS dedicado, geralmente nomeado de acordo com o componente e importado diretamente dentro dele. Isso facilita a manutenção ao associar o estilo diretamente à lógica do componente, mas os estilos podem vazar para outros componentes, já que o CSS tradicional não possui escopo isolado naturalmente. No exemplo abaixo apresento o arquivo css relacionado ao componente e sua respectiva importação no arquivo `MeuComponente.jsx`.

```css
/* MeuComponente.css */
.title {
  color: teal;
  font-size: 24px;
}
```

```jsx
// MeuComponente.jsx
import React from 'react';
import './MeuComponente.css';

function MeuComponente() {
  return <h2 className="title">Título do Componente</h2>;
}

export default MeuComponente;
```

CSS Modules

Para evitar o vazamento de estilos, os CSS Modules oferecem uma maneira de escopar classes de maneira local por padrão. Com esta abordagem, os nomes de classe são processados como hashes únicos no build, evitando conflitos e garantindo que os estilos de um componente não afetem outros. Para isso salve o arquivo com a extensão `.module.css`, exemplo abaixo:

```css
/* MeuComponente.module.css */
.title {
  color: teal;
  font-size: 24px;
}
```

A importação do arquivo de estilos e a aplicação da `ClassName` muda em relação ao css tradicional, a importação você deve fazer da seguinte forma:

```jsx
import styles from './MeuComponente.module.css';
```

E a aplicação da classe usando o styles importado seguido do nome da classe:

```
<h2 className={styles.title}>Título do Componente</h2>
```

Segue o exemplo completo abaixo:

```
// MeuComponente.jsx
import React from 'react';
import styles from './MeuComponente.module.css';

function MeuComponente() {
  return <h2 className={styles.title}>Título do Componente</h2>;
}

export default MeuComponente;
```

CSS Inline

Estilos inline em React são aplicados diretamente nos elementos com o atributo style, que aceita um objeto JavaScript. Embora os estilos inline sejam poderosos para aplicar estilos dinâmicos rapidamente, eles não suportam pseudoclasses como `:hover` ou `:active`. Além disso, são de difícil manutenção, pois para cada alteração, você deve localizar a aplicação do estilo em todos os elementos do seu projeto. Use esta maneira de forma pontual, quando você tiver certeza de que o estilo não vai mudar constantemente. Outra observação é que nas propriedades do estilo não pode ser usado o traço "-" para separar as palavras, mas sim em camelCase (`font-size` deve ser escrita como `fontSize`). Segue abaixo um exemplo:

```
// MeuComponente.jsx
import React from 'react';

function MeuComponente() {

  return <h2 style={{color:'purple', fontSize: '24px'} }>Título do
Componente</h2>;
}

export default MeuComponente;
```

Classes Dinâmicas

React permite aplicar classes de forma dinâmica usando expressões JavaScript. Isso é útil para alterar estilos com base no estado do componente ou em eventos do usuário, manipulando a renderização condicional das classes para fornecer respostas visuais interativas. Segue abaixo um exemplo que muda o estilo do elemento com base na classe que é atualizada via useState `isActive`, quando o usuário clica no botão a state é atualizada e consequentemente a classe e estilo são alterados.

```css
/* MeuComponente.css */
.active {
  color: red;
}

.inactive {
  color: grey;
}
```

```jsx
// MeuComponente.jsx
import React, { useState } from 'react';
import './MeuComponente.css';

function MeuComponente() {
  const [isActive, setIsActive] = useState(false);

  return (
    <h2
      className={isActive ? 'active' : 'inactive'}
      onClick={() => setIsActive(!isActive)}
    >
      Título do Componente
    </h2>
  );
}

export default MeuComponente;
```

Renderização de Listas no JSX e a Propriedade Key

No React, ao renderizar listas de elementos, a propriedade `key` é crucial para ajudar o React a identificar quais itens mudaram, foram adicionados ou removidos. Isso melhora o desempenho da aplicação ao permitir atualizações eficientes do DOM. Cada chave deve ser única entre os irmãos, ajudando o React a evitar renderizações desnecessárias e a manter a

consistência dos componentes. Abaixo, segue um exemplo de como renderizar listas em React. Utilizamos o método `map` do `JavaScript` para iterar sobre a lista, atribuindo uma propriedade `key` única a cada item. Isso garante que o React gerencie as atualizações de forma eficiente.

```
import React, { useState } from 'react';

function ListaExemplo() {
  const [itens, setItens] = useState(['Maça', 'Banana', 'Laranja']);

  return (
    <ul>
      {itens.map((item, index) => (
        <li key={index}>{item}</li>
      ))}
    </ul>
  );
}

export default ListaExemplo;
```

Conceito de Previous State

No React, o conceito de `Previous State` refere-se à utilização do estado anterior de um componente ao atualizar seu estado atual. Isso é particularmente útil em operações onde o novo estado depende do estado anterior, garantindo que as atualizações sejam seguras e consistentes, mesmo quando múltiplas atualizações ocorrem rapidamente.

```
import React, { useState } from 'react';

function ListaExemplo() {
  const [itens, setItens] = useState(['Maçã', 'Banana', 'Laranja']);

  const removerUltimoItem = () => {
    setItens((prevItens) => prevItens.slice(0, -1));
  };

  return (
    <div>
      <ul>
        {itens.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
      <button onClick={removerUltimoItem}>Remover Último Item</button>
    </div>
```

```
  );
}

export default ListaExemplo;
```

No exemplo acima, a função `removerUltimoItem` utiliza o estado anterior `prevItens` com `setItens((prevItens) => prevItens.slice(0, -1))`. Isso cria um novo array sem o último item, garantindo que a atualização do estado seja baseada no valor atual da lista. A função `slice(0, -1)` é usada para remover o último elemento da lista.

Renderização Condicional

Em React, a renderização condicional permite que você escolha qual conteúdo exibir com base em condições avaliadas durante a execução. Os operadores lógicos `&&` e `?:` (ternário) são comumente usados para esses casos.

Operador &&

O operador `&&` é usado para renderizar um elemento apenas se a condição for verdadeira. Se a condição antes do `&&` for avaliada como `true`, o JSX após o `&&` será renderizado. Isso é útil para exibir elementos opcionalmente sem usar `else`.

Operador Ternário ?:

O operador ternário `?:` é uma maneira compacta de realizar uma verificação `if-else` em linha. Ele permite escolher entre dois elementos baseados em uma condição: se a condição for verdadeira, o primeiro elemento é renderizado; caso contrário, o segundo elemento é exibido. Abaixo, segue um exemplo de componente usando ambos operadores.

```
import React, { useState } from 'react';

function RenderizacaoCondicional() {
  const [mostrarTexto, setMostrarTexto] = useState(true);
  const [numero, setNumero] = useState(1);

  return (
    <div>
      <h1>Exemplo de Renderização Condicional</h1>
      <button onClick={() => setMostrarTexto(!mostrarTexto)}>
        Alternar Texto
      </button>
          {mostrarTexto && <p>Este texto é exibido se mostrarTexto for
verdadeiro.</p>}
```

```
            <button onClick={() => setNumero(numero + 1)}>Incrementar
Número</button>
      <p>
        O número é {numero % 2 === 0 ? 'par' : 'ímpar'}.
      </p>
    </div>
  );
}

export default RenderizacaoCondicional;
```

Props em React

As `props` (propriedades) são uma forma de passar dados de um componente pai para um componente filho em React. Elas permitem que os componentes sejam reutilizáveis e configuráveis conforme necessário. Props são lidas pelos componentes filhos e, por definição, são imutáveis, o que significa que os componentes filhos não devem alterá-las diretamente. Essa passagem de parâmetros permite que componentes funcionem como funções no `JavaScript`, onde recebem argumentos e retornam um JSX configurado com base nesses valores.

O uso de props é fundamental para a construção de interfaces dinâmicas e modulares. Ao passar dados e funções como props, você consegue compor componentes complexos a partir de peças menores e mais simples, mantendo a lógica de apresentação separada e fácil de gerenciar.

Abaixo segue um exemplo comum de componente usando `props`.

```
function Aluno(props) {
  return (
    <div>
        <ul>
            <li>Nome: {props.nome}</li>
            <li>Curso: {props.curso}</li>
            <li>Turma: {props.turma}</li>
        </ul>
    </div>
  )
}

export default Aluno
```

Para adicionar o componente no `App.jsx` você deve passar os atributos da `prop` para que o componente possa receber estes valores e renderizar posteriormente. Segue abaixo o exemplo de chamada do componente `Aluno`.

```
function App() {

  return (
    <>
      <Aluno nome="Eduardo Gomes" curso="React" turma="Turma 01"/>
    </>
  )
}
export default App
```

Usando Destructuring

Podemos também utilizar a técnica de destruição (destructuring) de objetos para acessar a `prop` diretamente, tornando o código mais limpo. Conforme exemplo abaixo:

```
function Aluno({nome, curso, turma}) {
  return (
    <div>
        <ul>
            <li>Nome: {nome}</li>
            <li>Curso: {curso}</li>
            <li>Turma: {turma}</li>
        </ul>
    </div>
  )
}

export default Aluno
```

Usando uma lista para renderizar o componente

A partir de uma lista e de um componente que recebe uma `prop`, é possível renderizar o mesmo componente várias vezes com resultados diferentes em cada instância. Abaixo está um exemplo de como chamar o componente utilizando o método `map` para iterar sobre uma lista.

```
import Aluno from './components/Aluno'
function App() {
  const alunos = [
    {nome: "Eduardo", curso: "React", turma: "Turma 01"},
    {nome: "Maria", curso: "Java", turma: "Turma 02"},
    {nome: "Carlos", curso: "Javascript", turma: "Turma 03"},
```

```
    {nome: "José", curso: "Python", turma: "Turma 04"},
  ]

  return (
    <>
   {alunos.map((aluno, index) => (
        <Aluno key={index} props={aluno} />
      ))}
    </>
  )
}
export default App
```

Importante: Para funcionar corretamente é preciso receber no componente a prop utilizando a técnica de destruição, ou seja com `{}` no parâmetro de entrada, passando de `props` para `{props}`. Exemplo abaixo:

```
function Aluno({props}) {
  return (
    <div>
        <ul>
            <li>Nome: {props.nome}</li>
            <li>Curso: {props.curso}</li>
            <li>Turma: {props.turma}</li>
        </ul>
    </div>
  )
}

export default Aluno
```

Prop children

A prop `children` em React é uma propriedade especial que permite passar elementos ou componentes como **filhos** dentro de outros componentes. Isso é extremamente útil para criar componentes que podem encapsular outros conteúdos de forma flexível e reutilizável, semelhante a como elementos `HTML` podem conter outros elementos. A prop `children` pode conter tudo, desde strings e elementos `JSX` até outros componentes React, permitindo uma composição de interface mais modular e dinâmica.

Essa técnica é amplamente usada para componentes que devem ser altamente reutilizáveis e configuráveis em diferentes cenários. Abaixo segue um exemplo de utilização de prop children.

```
function Caixa({ children }) {
```

```
  return (
    <div>
      <h2>Exemplo de prop children</h2>
      {children}
    </div>
  );
}

export default Caixa
```

Abaixo apresento como este componente deve ser instanciado, neste caso o componente `Caixa` é usado para encapsular um título, um parágrafo e um botão. Esses elementos são passados automaticamente para a `prop children` do componente `Caixa`, que os renderiza dentro da div.

```
import Caixa from './components/Caixa'

function App() {
  return (
    <>
    <Caixa>
      <h3>Informação Importante</h3>
        <p>Este componente mostra como usar a prop <code>children</code> para
adicionar conteúdo dinâmico.</p>
      <button onClick={() => alert('Botão clicado!')}>Clique aqui</button>
    </Caixa>
    </>
  )
}

export default App
```

Função como Prop

Passar funções como props em React é uma prática comum que permite que componentes filhos comuniquem alterações ou eventos de volta aos componentes pais. Essa técnica é essencial para gerenciar o fluxo de dados unidirecional que o React favorece. Ao passar uma função como prop, o componente filho pode chamar essa função para enviar dados de volta para o pai, desencadear ações, ou modificar o estado do componente pai. Isso torna a interação entre componentes mais dinâmica e mantém a lógica do componente modulares e organizadas. Segue exemplo abaixo:

```
function BotaoIncrementar({ funcao }) {
  return (
  <button onClick={funcao}>Incrementar</button>
```

```
);
}

export default BotaoIncrementar
```

O exemplo abaixo apresenta como a função é enviada como prop para o componente filho.

```
import { useState } from 'react'
function App() {
  const [contador, setContador] = useState(0);

  const incrementarContador = () => {
    setContador(contador + 1);
  };

  return (
    <div>
      <h1>Contador: {contador}</h1>
      <BotaoIncrementar funcao={incrementarContador} />
    </div>
  );
}

export default App;
```

Explicação:

- Função `incrementarContador`: Definida no componente pai App, ela é responsável por atualizar o estado contador.
- Passagem da Função: A função é passada como prop `funcao` para o componente filho `BotaoIncrementar`.
- Chamada no Filho: O botão no componente `BotaoIncrementar` chama `funcao` quando clicado, incrementando assim o contador no componente pai.

Essa estratégia permite o controle e atualização do estado em um componente pai através de interações em componentes filhos.

Requisições HTTP com React

Requisições HTTP são essenciais em aplicações React para interagir com servidores, recuperando ou enviando dados. Elas permitem que o aplicativo se comunique com APIs para obter recursos atualizados, enviar informações do usuário, e muito mais. Os verbos HTTP definem a ação que está sendo realizada e são fundamentais para construir uma aplicação dinâmica e conectada.

GET

O verbo GET é utilizado para recuperar dados de um servidor. É a operação mais comum e serve para trazer informações sem alterar o estado do recurso.

POST

POST é utilizado para enviar dados ao servidor, geralmente para criar um novo recurso. Diferente do GET, ele pode alterar o estado no servidor.

PUT

PUT é usado para atualizar um recurso existente no servidor. Ele substitui o recurso atual pelos dados enviados.

DELETE

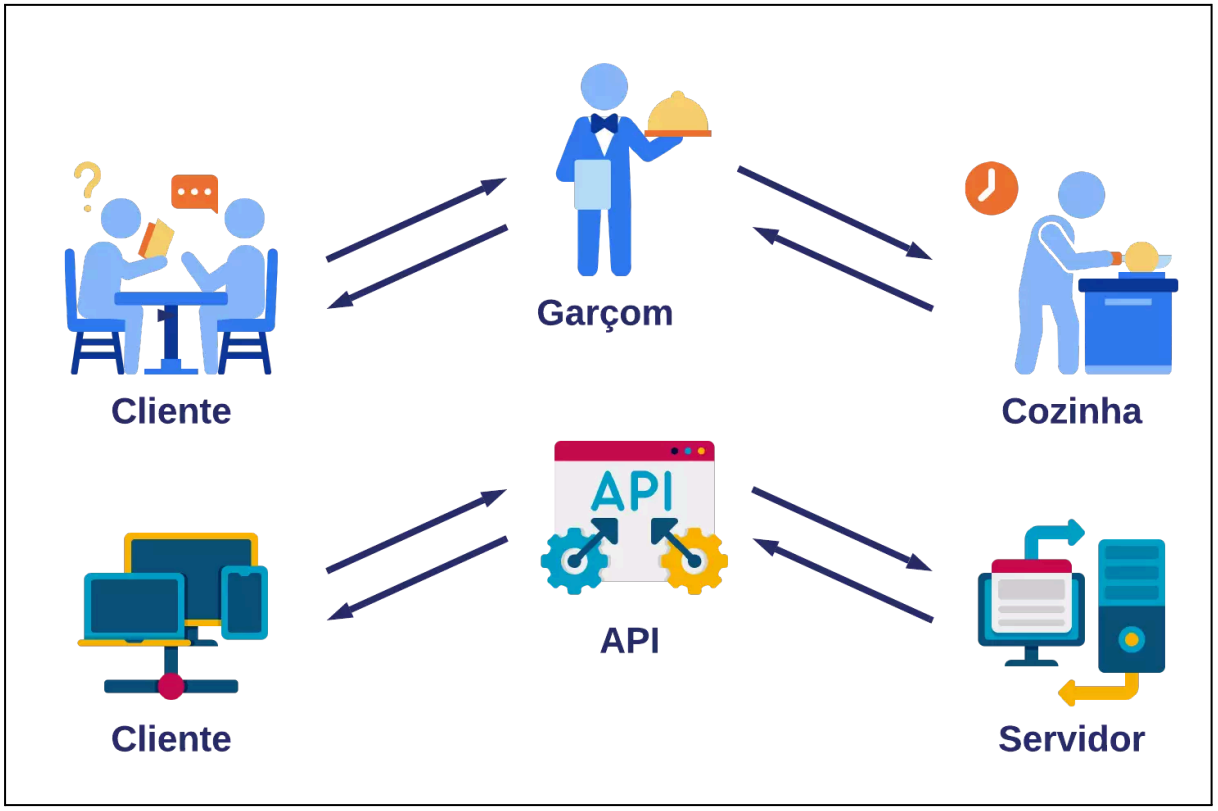
DELETE é utilizado para remover um recurso do servidor. Ele é direto e geralmente não requer um corpo na requisição.

PATCH

O verbo PATCH é utilizado para aplicar modificações parciais a um recurso existente no servidor. Ao contrário do PUT, que substitui completamente o recurso, o PATCH permite que apenas as mudanças específicas sejam enviadas, tornando-o mais eficiente para atualizações parciais. É ideal para cenários onde apenas alguns campos de um objeto precisam ser alterados, minimizando a quantidade de dados trafegados e o risco de sobrescrever inadvertidamente dados não relacionados.

O que é uma API?

Uma **API** (Application Programming Interface, ou Interface de Programação de Aplicações) é um conjunto de regras, protocolos e ferramentas que permite a **comunicação entre diferentes sistemas**, aplicações ou componentes de software. Em outras palavras, uma API define como as funcionalidades de um sistema podem ser utilizadas por outros programas, de forma padronizada e segura, sem que seja necessário conhecer toda a sua implementação interna. As APIs facilitam a integração, a reutilização de recursos e o desenvolvimento mais rápido de softwares, já que permitem que diferentes aplicações "conversem" entre si.



Hook useEffect

O useEffect é um hook em React que permite realizar efeitos colaterais em componentes funcionais. É usado para operações como buscar dados de APIs, subscrição a eventos, ou modificar diretamente o DOM. Executa-se após a renderização do componente, garantindo que qualquer ação que precise interagir com o elemento completo esteja sincronizada.

Como Funciona

Efeitos sem Dependências: Executa após cada renderização. É configurado sem a array de dependências.

```
useEffect(() => {
  console.log('Executa após cada renderização');
});
```

Efeitos com Dependências: Executa apenas quando as dependências especificadas mudam. No exemplo abaixo, quando a state `dados` mudar, será executado todo o bloco de comandos dentro da `useEffect`.

```
useEffect(() => {
  console.log('Executa quando `dados` mudar');
}, [dados]);
```

Efeitos de Montagem e Desmontagem: Quando a array de dependências é vazia, executa apenas na montagem e desmontagem. Útil para inicializações e limpezas.

```
useEffect(() => {
  console.log('Executa na montagem');

  return () => {
    console.log('Executa na desmontagem');
  };
}, []);
```

Exemplo de Uso: Suponha que você deseje buscar dados de uma API quando um componente for renderizado inicialmente:

```
import React, { useEffect, useState } from 'react';

function ListaDeUsuarios() {
  const [usuarios, setUsuarios] = useState([]);

  useEffect(() => {
    fetch('https://api.exemplo.com/usuarios')
      .then(response => response.json())
      .then(data => setUsuarios(data));
```

```
  }, []); // Array vazia significa que executa apenas uma vez, na montagem

  return (
    <ul>
      {usuarios.map(usuario => (
        <li key={usuario.id}>{usuario.nome}</li>
      ))}
    </ul>
  );
}

export default ListaDeUsuarios;
```

JSON Server

O JSON Server é uma ferramenta que serve para simular uma API rapidamente, utilizando um simples arquivo JSON como banco de dados. Ele é útil para desenvolvimento e testes, permitindo que desenvolvedores criem um backend completo sem escrever código do lado servidor. O JSON Server lê e escreve dados no arquivo JSON, proporcionando um conjunto de **endpoints** que permite operações **CRUD** (Create, Read, Update, Delete), agilizando a prototipagem e desenvolvimento frontend.

Passos para Instalação

Instale o JSON Server globalmente usando o npm:

```
npm install -g json-server
```

Exemplo de um arquivo de banco de dados JSON: Crie uma pasta dentro do projeto chamada `./data` e dentro desta pasta crie um arquivo chamado `db.json` com o seguinte conteúdo:

```
{
  "tasks": [
    {
      "id": "1",
      "text": "Ler um capítulo de um livro",
      "done": true
    },
    {
      "id": "2",
      "text": "Estudar React e Javascript",
      "done": false
    }
  ]
}
```

Inicie o JSON Server para servir seus dados. No terminal digite o seguinte comando:

```
json-server --watch ./data/db.json --port 3001
```

Com isso, o JSON Server estará funcionando em `http://localhost:3001`, permitindo que você realize operações sobre suas "tasks" (tarefas), usando `endpoints` como /tasks, simplificando assim o processo de desenvolvimento do frontend, sem precisar desenvolver uma API.

Hook useRef

O `useRef` é um hook do React que permite criar uma referência mutável que persiste durante o ciclo de vida completo do componente. Diferente de `useState`, alterar um objeto `useRef` não causa re-renderização do componente. Ele é amplamente utilizado para acessar diretamente elementos DOM, armazenar valores mutáveis sem causar novas renderizações.

Exemplos Comuns de Uso

- **Acessar o DOM diretamente:**
  - O useRef pode ser utilizado para criar uma referência a um elemento DOM, permitindo manipulações diretas, como focar um input ou rolar até uma determinada posição.
- **Armazenar valores mutáveis:**
  - Serve para manter valores que precisam ser persistidos entre renderizações mas que não necessitam causar re-renderizações quando mudam.

Exemplo de Utilização

Vamos criar um exemplo onde um campo de entrada é automaticamente focado quando o componente é montado:

```
import React, { useRef, useEffect } from 'react';

function CampoTexto() {
  const inputRef = useRef(null);

  useEffect(() => {
    // Focar o campo de entrada quando o componente for montado
    inputRef.current.focus();
  }, []);

  return (
    <div>
      <input ref={inputRef} type="text" placeholder="Digite algo..." />
    </div>
  );
}

export default CampoTexto;
```

Explicação

- **Criação de Referência:** `const inputRef = useRef(null);` cria uma referência que será associada ao campo de entrada.
- **Foco no Elemento:** Com `useEffect`, chamamos `inputRef.current.focus();` para focar automaticamente o campo de entrada quando o componente é montado, melhorando a usabilidade da interface.
- **Acesso ao DOM:** `inputRef.current` fornece acesso direto ao elemento DOM, permitindo manipulações ou obtenção de informações conforme necessário.

O `useRef` é uma ferramenta poderosa para gerenciar referências e manipulações diretas, mantendo o comportamento funcional e a organização limpa dentro dos componentes React.

Publicando uma Aplicação no GitHub Pages

Publicar sua aplicação no GitHub é uma excelente maneira de compartilhar seu projeto, colaborar com outras pessoas e gerenciar o histórico do seu código. Aqui está um guia passo a passo, desde a instalação do Git até o envio de um projeto criado com Vite.

Instalação e Configuração do Git

O Git é a ferramenta que permite versionar seu código localmente e interagir com o GitHub.

Baixar e Instalar:

Acesse o site oficial do Git em https://git-scm.com/ e baixe o instalador para o seu sistema operacional (Windows, macOS ou Linux). Siga as instruções do instalador para finalizar a instalação.

Configuração Inicial:

Após a instalação, abra o terminal ou prompt de comando e configure seu nome de usuário e e-mail. Isso ajudará a identificar suas contribuições nos seus projetos.

```
git config --global user.name "Seu Nome"
git config --global user.email "seuemail@example.com"
```

Preparar para Produção:

Para que os links, imagens e arquivos CSS da sua aplicação funcionem corretamente no GitHub Pages, você precisa definir o caminho base da aplicação no arquivo `vite.config.js`.

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

// https://vite.dev/config/
export default defineConfig({
  plugins: [react()],
    base:  "/nome-do-seu-repositorio/",  //  Substitua  com  o  nome  do  seu
repositório
})
```

O comando de build do Vite otimiza sua aplicação, gerando os arquivos estáticos prontos para serem publicados. Os arquivos serão criados na pasta `dist`.

```
npm run build
```

O GitHub Pages, nativamente, só permite como pasta base da publicação o diretório `raiz / (root)` ou uma pasta chamada **/docs** localizada na raiz do repositório. Por esse motivo, antes de enviar o projeto para o repositório e realizar o deploy, é necessário renomear a pasta **`dist`** para **`docs`**, garantindo assim que o conteúdo seja corretamente disponibilizado pela plataforma durante a configuração do site.

Enviando a Aplicação para o GitHub

Agora que sua aplicação está pronta, é hora de enviá-la para um repositório no GitHub.

Inicializar um Repositório Local:

Dentro da pasta do seu projeto (meu-projeto), inicialize um repositório Git local.

```
git init
```

Adicionar e Confirmar os Arquivos:

Adicione todos os arquivos do projeto ao Git e crie seu primeiro commit. O commit funciona como um "ponto de salvamento" do seu projeto.

```
git add .
git commit -m "Primeiro commit"
```

Criar um Repositório no GitHub:

Acesse github.com e crie um novo repositório. Dê um nome a ele (o mesmo nome usado no arquivo `vite.config.js`) e, se desejar, adicione uma descrição. Não é necessário inicializar com um README.md ou outros arquivos.

Conectar o Repositório Local ao GitHub:

No terminal, adicione a URL do repositório que você acabou de criar no GitHub. Substitua usuario/nome-do-repositorio pelos dados corretos do seu repositório.

```
git remote add origin https://github.com/usuario/nome-do-repositorio.git
```

Enviar os Arquivos:

Envie o código da sua branch local (main) para o repositório remoto no GitHub.

```
git branch -M main
git push -u origin main
```

Pronto! Sua aplicação está agora disponível no GitHub, pronta para ser compartilhada, clonada e colaborada.

Visualizando a Aplicação com GitHub Pages

Para que a sua aplicação seja visualizada publicamente através do GitHub Pages, é necessário ativá-lo nas configurações do seu repositório.

Ativar o GitHub Pages:

1. Vá para a página do seu repositório no GitHub e clique na aba **`Settings`** (Configurações).
2. No menu lateral, clique em **`Pages`**.
3. Em **`Build and deployment`**, na opção `Source`, selecione **`Deploy from a branch`**.
4. Em **`Branch`**, escolha sua branch principal (geralmente **`main`**), selecione a pasta **`docs`** e clique em **`Save`**.

O GitHub Pages levará alguns minutos para compilar e publicar sua aplicação. Após o processo, um link para a sua página estará disponível na mesma tela.

Trabalho Final: Projeto de Frontend em React

O objetivo deste trabalho é desafiar os alunos a aplicar os conhecimentos adquiridos ao longo do curso, desenvolvendo um projeto de frontend utilizando React e JavaScript. Este projeto servirá como uma oportunidade para cada aluno explorar sua criatividade e habilidades técnicas, enquanto cria uma aplicação prática e funcional.

Descrição do Trabalho

- **Escolha do Projeto:** Cada aluno deve selecionar um projeto de front que seja desafiador, mas realizável dentro do tempo disponível. O projeto deve estimular o aprendizado e permitir o desenvolvimento de novas competências.
- **Desenvolvimento:** Utilize React com JavaScript para criar a aplicação. Escolha componentes e funcionalidades que demonstrem o uso eficaz de conceitos aprendidos, como, criação de componentes, useState, useEffect e integração com APIs.
- **Apresentação em Vídeo:** Após completar o projeto, grave um vídeo de 3 a 6 minutos. A apresentação deve incluir:
    - Visão geral do projeto e seus objetivos.
    - Demonstração dos principais componentes e suas funcionalidades.
    - Reflexão sobre os desafios enfrentados e como foram superados.
- **Submissão:** Após a gravação, salve o vídeo em uma plataforma de armazenamento como Google Drive ou YouTube. Publique o link do vídeo no Moodle para avaliação.

Critérios de Escolha

- **Desafio Adequado:** Escolha um projeto que não seja demasiadamente fácil, mas também não tão complexo que não possa ser concluído no tempo disponível. A palavra é: **Desafie-se!**
- **Deadline:** O prazo para entrega é **30/01/2026**.